소프트웨어 안정성의 한계와 해결방안

박정웅

Abstract

수학은 결정 가능성을 만족하지 못한다는 결점이 있다. 이는 소프트웨어 설계에도 적용될 수 있고, 그 어떠한 알고리즘도 소프트웨어가 버그를 가지는지 완벽하게 확인할 수 없다. 하지만 소프트웨어의 버그는 수많은 문제를 일으킬 수 있으으로 최소한 완벽한 소프트웨어를 근사할 수는 있어야 한다. 이를 위해 소프트웨어의 간소화, 여러 프로그래밍 언어의 개념들 및 테스팅 기법 등을 사용할 수 있을 것이다.

수학은 완전성(complete), 무모순성(consistent), 결정 가능성(decidability) 중 어느 하나도 만족하지 못한다는 결점이 있다. 예로부터 수학자들은 여러 난제를 해결하기 위하여 부단히 노력하였다. 그 중 페르마의 마지막 정리와 같이 수백 년 동안 풀리지 않던 난제가 해결되는가 하면, 리만 가설처럼 여러 수학자를 절망에 빠트린 문제도 있다. 심지어 간단해 보이는 쌍둥이 소수 문제조차 어떠한 해답도 나오지 않은 상태이다. 이러한 난제들을 보면 항상 한 가지 의문이 들었다. 이렇게나 많은 수학자가 오랫동안 도전해도 해결할 수 없는 난제들은 과연 증명될 수 있기는 한 문제들인가? 어쩌면 증명할 수 없는 문제들을 붙잡고 있는 것이 아닌가. 이 의문에 대하여 괴델은 불완전성 논리로 수학의 완전성과 무모순성을 부정하였고, 튜링은 그의 튜링 머신을 통하여 결정 가능성을 부정하였다. 즉, 그 어떠한 수 체계를 정립하여도 쌍둥이 소수와 같은 난제들의 증명 방법은 없을 수가 있고 어떤 명제가 사실인지 알 수 있는 효과적인 알고리즘이 없다는 뜻이다. 이러한 수학의 결점은 매우 큰 혼동을 초래한다. 어떠한 명제를 증명할 수 있는 진리가 존재하지 않을 수도 있다는데 그 누가 수백 년간 풀리지 않는 난제에 도전해 볼 엄두가 나겠는가.

소프트웨어의 안정성 역시 수학이 가지는 결점을 공유한다. 즉, 어떠한 소프트웨어가 오류로부터 자유로운지 확인할 수 있는 완벽한 알고리즘은 존재하지 않는다. 소프트웨어는 태생적으로 오류에 취약하다. 작게는 소프트웨어 내부의 제로 디비전 문제, 데드락, 널 포인터 접근부터 크게는 메모리 관리 문제까지, 소프트웨어는 항상 여러 종류의 버그로부터 공격받는다. 문제는 소프트웨어 버그가 너무나 큰 결과를 야기할 수 있다는 점이다. 인스타그램의 서버 오류는 분당 수억 원의 재정 손해를 유발하고, 여객기에 탑재되는 소프트웨어에서의 커널 패닉은 수백 명의 사상자로 직결된다. 따라서 소프트웨어 설계의 핵심은 버그로부터 자유로운 프로그램이라 생각한다. 이를 위해선 어떠한 소프트웨어가 버그를 가지고 있는지 없는지 확인할 수 있는 알고리즘이 필요하다. 아쉽게도 이는 수학의 결정 가능성(decidability)문제로 치환될 수 있고, 정지 문제(halt problem)를 통하여 그러한 알고리즘은 존재할 수 없음이 증명되었다. 수학에서의 혼돈이 소프트웨어의 안정성 문제에까지 영향을 미친 것이다. 하지만 수학에서의 혼돈과 달리 소프트웨어에서의 혼돈은 지금 당장 해결되어야 한다. 여객기의 커널 패닉으로 수백 명이 다치는 것을 보고만 있을 수는 없지 않은가.

그럼 소프트웨어의 안정성 문제는 어떻게 극복되어야 하는가? 소프트웨어가 버그를 가졌는지 확인할 수 있는 완벽한 알고리즘은 이론상 존재하지 않지만, 소프트웨어가 최대한 버그를 가지고 있지 않게 완벽한 소프트웨어를 근사할 수는 있을 것이다. 가장 직관적인 방법은 소프트웨어의 크기와 복잡성을 줄이는 방법이 떠오른다. 화성 탐사선은 윈도우와 같이 방대한 운영체제를 필요하지 않는다. 화성 탐사 임무 중 블루스크린이라도 발생하는 순간 막대한 양의 재정적 손실이 발생하기 때문이다. 따라서 주어진 작업만 할 수 있는 가장 단순한 소프트웨어를 개발하는 방법론이 필요하다. 하지만 근래에는 소프트웨어의 크기 자체가 매우 방대해지고 있기 때문에 위 방식은 실용적이지 않아 보인다. 또 다른 방법은 현대에 연구된 여러 가지 프로그래밍 언어의 개념을 적극적으로 차용하여 소프트웨어를 설계하는 방법이다. 비교적 과거의 언어인 C나 C++ 등은 흔히 수십억 달러의 실수(billion dollar mistake)라 일컬어지는 널 포인터 접근 오류를 가지고 있다. 하지만 러스트(rust)는 언어 차원에서 널 포인터 개념을 사용하지 않고 소유권과 수명이란 개념을 사용하여 메모리 안전성을 꾀한다. 이 외에도 현대의 언어들은 적극적인 타입 시스템을 사용하여 소프트웨어의 버그를 줄이려 노력한다. 마지막 방법으론, 결국 소프트웨어 버그를 찾아내는 테스팅 기법이 가장 중요하다고 생각된다. 비록 소프트웨어 내에서 발생하는

모든 버그를 잡아낼 수는 없지만, 현재 사용되는 다양한 소프트웨어의 테스팅 기법들이 대부분의 버그는 잡을 수 있다. 따라서 소프트웨어의 설계를 함에 있어 소프트웨어 테스팅 기법이 동시에 수반되어야 할 것이다.